Solving Atomic Broadcast with Indirect Consensus

Richard Ekwall
nilsrichard.ekwall@epfl.ch

André Schiper
andre.schiper@epfl.ch

École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland

Abstract

In previous work, it has been shown how to solve atomic broadcast by reduction to consensus on messages. While this solution is theoretically correct, it has its limitations in practice, since executing consensus on large messages can quickly saturate the system. The problem can be addressed by executing consensus on message identifiers instead of the full messages, in order to decouple the size of the messages from the size of the data sent by the consensus algorithm.

In this paper, we study the impact of executing consensus on message identifiers instead of on the full messages, in the context of solving atomic broadcast. We also discuss the implications of executing consensus on message identifiers on the consensus and atomic broadcast algorithms.

1 Introduction

1.1 Context

Atomic broadcast (or total order broadcast) and consensus are important abstractions in fault tolerant distributed computing. Atomic broadcast ensures that messages that are sent are delivered in the same order by all processes [6]. Consensus allows a group of processes to reach a common decision. In [2], the authors present a reduction of atomic broadcast to consensus. In this reduction, the atomic broadcast algorithm performs consensus runs on sets of messages in order to determine the delivery order of those messages.

While this is correct from a theoretical point of view, it is inefficient in practice. Indeed, executing consensus on messages can lead to heavy network usage if the messages are large. Instead, if consensus is executed on message identifiers (*indirect* consensus), the messages themselves only need to be diffused once and the ordering process is then done on light-weight message identifiers.

Executing consensus on message identifiers has already been done in previous group communication stack implementations [4, 10] and has always been seen as being easy, given a consensus algorithm on messages. However, in these group communication implementations, the consensus algorithms were not adapted to handle message identifiers instead of messages. As a consequence, if at least one process can crash, it can lead to a faulty execution, as we show in this paper.

To correctly implement a group communication stack, the consensus and atomic broadcast algorithms need to be adapted to the case where the decision is taken on identifiers instead of messages. We show that these modifications are not trivial for all consensus algorithms and can affect their resilience.

1.2 Contributions

We start by discussing and illustrating the advantages of executing consensus *indirectly* on message identifiers rather than on messages. Two contributions are then presented in this paper: we (1) start by presenting *indirect* consensus, and show what guarantees it must provide to ensure the correctness of the atomic broadcast algorithm. We then (2) show that the transformation of failure-detector based consensus algorithms on messages into *indirect* consensus algorithms on message identifiers is far from trivial: the resilience (i.e. the number of supported failures) of some consensus algorithms can be affected by the modifications. To illustrate this, two $\Diamond\mathcal{S}$-based consensus algorithms are adapted into indirect consensus algorithms that work on message identifiers. The resilience of one of the algorithms is affected by the modifications whereas the other one isn't.

The paper is structured as follows. In Section 2, we motivate the use of consensus on message identifiers

rather than on messages and present the formal specification of *indirect* consensus. Section 3 illustrates the modifications that are needed to transform two consensus algorithms with the failure detector $\Diamond\mathcal{S}$ into indirect consensus algorithms. The section emphasizes the fact that not all consensus algorithms on messages can be trivially modified into indirect consensus algorithms on message identifiers. Section 4 compares the performance of the consensus and indirect consensus algorithms presented in Section 3. Finally, Section 5 concludes this paper.

2 Motivation and indirect consensus

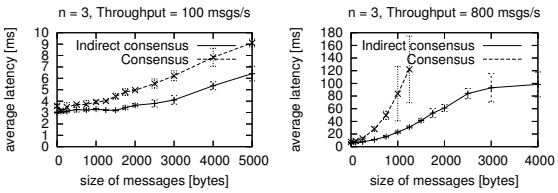
2.1 Atomic broadcast on message identifiers

In the following paragraphs, we discuss the use of message identifiers in atomic broadcast algorithms. We start by giving the specifications of reliable and atomic broadcast. We then recall the reduction of atomic broadcast to consensus and show performance comparisons between executions of atomic broadcast using messages and executions using message identifiers. The specifications and the short reminder on the reduction of atomic broadcast to consensus help in understanding the problems involved when executing consensus on message identifiers.

We consider an asynchronous system composed of n processes taken from a set $\Pi = \{p_1, \ldots, p_n\}$. The processes communicate by passing messages over reliable channels and can only fail by crashing (no Byzantine failures). A process that never crashes is said to be *correct*, otherwise it is *faulty*.

Informally, reliable broadcast guarantees that all correct processes deliver the same set of messages. Formally, reliable broadcast is defined by two primitives *rbroadcast* and *rdeliver* and satisfies three properties [6]: (1) *Validity:* If a correct process p *rbroadcasts* a message m, then it eventually *rdelivers* m, (2) *Uniform integrity:* For any message m, every process p *rdelivers* m at most once and only if m was previously *rbroadcast*, (3) *Agreement:* If a correct process *rdelivers* m, then all correct processes eventually *rdeliver* m.

(Uniform) atomic broadcast is reliable broadcast augmented with a uniform agreement property and a total order property. The (uniform) atomic broadcast problem is defined by two primitives *abroadcast* and *adeliver* that satisfy the (1) Validity and (2) Uniform integrity properties of reliable broadcast and the additional uniform agreement and order properties: (3) *Uniform Agreement:* If a process (correct or not) *adelivers* m, then all correct processes eventually *adeliver* m, and (4) *Uniform Total Order:* If some process, correct or faulty, *adelivers* m before m', then every process *adelivers* m' only after it has adelivered m.



(a) 100 msgs/s (b) 800 msgs/s

Figure 1. Latency of the atomic broadcast algorithm versus size of the messages in a system with 3 processes

In [2], the authors present a reduction of atomic broadcast to consensus. In this reduction, whenever a message m is *abroadcast*, m is reliably broadcast to all processes. Following this, whenever a process receives a message that it hasn't already *adelivered*, it executes consensus in order to reach a decision with the other processes on the next message to *adeliver*. In the reduction of atomic broadcast to consensus in [2], the processes thus execute a sequence of consensus runs on sets of messages as long as new messages are *abroadcast*.

This reduction is correct. However, in a system where the messages are *large*, the consensus runs are executed on sets of *large* messages. Thus, the size of the data exchanged by the consensus algorithm is large and can potentially saturate the system. In order to avoid this, consensus can be executed on message identifiers instead of the messages themselves. This decouples the size of the messages from the size of the data exchanged by consensus. Since the relationship between the messages and their identifiers is bijective, the delivery order of the messages can easily be inferred from the ordered sequence of message identifiers (which ensures the Uniform Total Order property of atomic broadcast).

The performance gain when using message identifiers instead of messages in consensus is not negligible. Indeed, the size of a message identifier is independent of the size of the message itself. Thus, the size of the data exchanged by consensus remains constant as the size of the messages increases. Figure 1 illustrates the performance difference between executing consensus on messages or message identifiers in the context of atomic broadcast. The performance metric for atomic broadcast is the *latency*, defined as the average (over all processes) of the elapsed time between *abroadcasting* a message m and *adelivering* m. The figure shows the latency of atomic broadcast as a function of the size of the messages. The

results are shown for two *throughputs* (the overall rate of atomic broadcasts in the system : 100 or 800 messages per second). The tests were done using the Neko framework on a local area network with Pentium III machines. More details on the framework and the system setup can be found in Section 4.

One can clearly see that as the size of the messages increases, the latency of consensus on message identifiers is lower than the latency when using entire messages. This result becomes clearer as the throughput of atomic broadcasts increases (and as the size of the system increases, as [3] shows). As a consequence, except for trivial conditions (low throughputs and small systems), executing consensus on message identifiers rather than on entire messages is clearly justified.

2.2 Violating validity of atomic broadcast

Executing consensus on message identifiers implies that the consensus and atomic broadcast algorithms must be adapted to explicitly handle message identifiers instead of messages, as we now show. More specifically, we show that if the atomic broadcast algorithm directly executes the original consensus algorithm in [2] on message identifiers, the validity property of atomic broadast could be violated. Imagine that a faulty process p reliably broadcasts a new message m and starts executing consensus on its message identifier $id(m)$. Let us assume that the consensus algorithm decides on $id(m)$, p crashes and no other process receives a copy of m. No other process than p is able to deliver m (and thus any message ordered after m). Furthermore, in order to guarantee the total order property of atomic broadcast, $id(m)$ cannot be removed from the sequence of ordered message identifiers. As a consequence, the validity property of atomic broadcast is violated in such an execution, since m and any following message in the ordered sequence cannot be delivered.

The problem described above could be avoided by using uniform reliable broadcast instead of reliable broadcast in the atomic broadcast algorithm. Uniform reliable broadcast guarantees that if at least one process (correct or not) delivers a message, then all correct processes eventually deliver that message [6]. Since the atomic broadcast algorithm in [2] only executes consensus on messages that have been uniformly reliably delivered, this solution guarantees that all correct processes eventually receive a copy of any message ordered by consensus. However, the cost of using uniform reliable broadcast is higher than that of reliable broadcast.

Instead of using uniform reliable broadcast and incurring its cost, we suggest to adapt the consensus algorithm to handle message identifiers and provide additional properties that ensure the correctness of atomic broadcast.

2.3 Indirect consensus

The motivation of introducing *indirect* consensus is to capture the differences between executing consensus on messages and on message identifiers. Instead of executing consensus directly on messages, we want to *indirectly* execute consensus on message identifiers. Simultaneously, indirect consensus has to offer guarantees to atomic broadcast so that all the messages whose identifiers have been ordered can be delivered by atomic broadcast.

In *indirect* consensus, each proposal is a pair (v, rcv), where v is a set of message identifiers (and $msgs(v)$ are the messages whose identifiers are in v). rcv is a function such that $rcv(v)$ returns true only if the process has received $msgs(v)$. Whenever a decision is taken on v, indirect consensus must ensure that all correct processes eventually receive $msgs(v)$. In the context of indirect consensus, we introduce the following hypothesis on the rcv function:
Hypothesis A: If $rcv(v)$ is true for a correct process, then $rcv(v)$ is eventually true for all correct processes.

Formally, we specify *indirect* consensus similarly to consensus, in terms of two primitives: *propose(v,rcv)* and *decide(v)*. The (uniform) *indirect* consensus problem is then specified by five properties. The four first properties are (almost) identical to (uniform) consensus:

> *Termination* : If the *Hypothesis A* holds, then every correct process eventually decides some value.
>
> *Uniform integrity* : Every process decides at most once.
>
> *Uniform agreement* : No two processes (correct or not) decide a different value.
>
> *Uniform validity* : If a process decides v, then (v, rcv) was proposed by some process in Π.
>
> *No loss* : If a process decides v at time t, then one correct process has received $msgs(v)$ at time t.

The No loss property implies that indirect consensus has to be able to know if given v, the messages $msgs(v)$ have been received. This information is provided by the rcv function (the function would typically be provided by the atomic broadcast algorithm).

2.4 Reducing atomic broadcast to indirect consensus

The reduction of atomic broadcast to indirect consensus is almost identical to the reduction of atomic broad-

Algorithm 1 Skeleton of the atomic broadcast algorithm using message identifiers

```
1: Initialisation:
2:     received_p ← ∅        {set of messages received by process p}
3:     unordered_p ← ∅  {set of identifiers of messages received but
       not yet ordered by process p}
4:     ...
5: procedure A-broadcast(m)        {To A-broadcast a message m}
6:     R-broadcast message(m) to all
7: procedure rcv(ids)
8:     return true if msgs(ids) ⊆ received_p, false otherwise.
9: when R-deliver message(m)
10:    received_p ← received_p ∪ {m}
11:    if id(m) has not been ordered then
12:        unordered_p ← unordered_p ∪ {id(m)}
13: when unordered_p ≠ ∅  {unordered messages⇒run consensus}
14:    propose(unordered_p, rcv)
15:    wait until decide(idSet)
16:    unordered_p ← unordered_p \ idSet
```

cast to consensus in [2]. The main difference resides in the fact that instead of executing consensus on a set of messages, indirect consensus is executed on a pair *(set of message identifiers, rcv function)*. The validity of atomic broadcast is ensured by the No loss property of indirect consensus: the messages in $msgs(v)$ corresponding to the decision v of indirect consensus are *rdelivered* by at least one correct process (and thus all correct processes eventually *rdeliver* $msgs(v)$).

Since the modifications to atomic broadcast are relatively minor, only a skeleton of the algorithm is shown in Algorithm 1 (the complete algorithm is in [3]). This skeleton is used later to prove properties of the indirect consensus algorithms. It shows the following: whenever *abroadcast* is called on a message m, then m is *rbroadcast* to all processes (line 6). If a process *rdelivers* a message that hasn't been ordered yet (line 12), consensus is executed on the unordered message identifiers (lines 13 to 16). The rcv function is shown in lines 7 and 8 of Algorithm 1.

We now show that the rcv function of atomic broadcast satisfies the *Hypothesis A* above. If $rcv(v)$ is true for a correct process, then all messages $msgs(v)$ whose identifier is in v have been previously *rdelivered*. Following the Agreement property of reliable broadcast, all correct processes eventually *rdeliver* $msgs(v)$ and thus $rcv(v)$ is eventually true for all correct processes.

3 Solving indirect consensus

We now show how to solve indirect consensus. We start by discussing what properties an indirect consensus algorithm must enforce in order to guarantee the No loss property. Two consensus algorithms are then adapted into indirect consensus algorithms: (1) the Chandra-Toueg $\Diamond\mathcal{S}$ consensus algorithm (*CT*) and (2) the Mostéfaoui-Raynal $\Diamond\mathcal{S}$ consensus algorithm (*MR*). These two algorithms illustrate two cases. CT illustrates the case of a consensus algorithm that is fairly easy to adapt into an indirect consensus algorithm; MR illustrates the case of a consensus algorithm whose resilience is reduced by the adaptation into an indirect consensus algorithm (the indirect consensus algorithm requires $\lceil \frac{2n+1}{3} \rceil$ correct processes where the original consensus algorithm required $\lceil \frac{n+1}{2} \rceil$ correct processes).

3.1 Conditions on the correctness of indirect consensus algorithms

We present the conditions that indirect consensus algorithms must fulfill in order to ensure the No loss property. To do this we introduce the two following definitions:

Definition: v-valent configuration. As in [5], we say that a configuration is v-*valent* at time t if any decision that is taken after t can only be v. As an example, consider a configuration where all processes start consensus at time t_0 with the same initial value v. Such a configuration is v-valent at time t_0 (although the first process to decide only does so after t_0).

Definition: v-stable configuration. We say that a configuration is v-*stable* at time t if $f+1$ processes have received $msgs(v)$ at time t (f is the maximum number of processes that may crash). v-stability ensures that at least one correct process has received $msgs(v)$.

From these definitions, if a configuration is v-valent or v-stable at time t, any configuration at time $t' > t$ is also v-valent, respectively v-stable.

Ensuring the No loss property We now show that for the No loss property to hold, it is necessary and sufficient that any configuration that is v-valent at some time t is also v-stable at t.

We first show that the No loss property of an algorithm holds, if the algorithm guarantees that a v-valent configuration is also v-stable. Let us assume that the first decision on a value v is taken at some time t_0. From the Uniform agreement property of the indirect consensus algorithm, the configuration is v-valent at time t_0. Since the v-valence of a configuration implies that the configuration is also v-stable, v-stability also holds at time t_0. Thus, the No loss property holds.

Now, we show that if an algorithm allows a v-valent configuration that is not v-stable, then the No loss property of the algorithm does not hold. Let us assume that the system reaches a v-valent configuration at time t that

is not v-stable. Since the configuration is not v-stable, at most f processes have received $msgs(v)$ at time t. If those f processes crash, all copies of $msgs(v)$ are lost and no correct process ever receives $msgs(v)$. Either no decision is taken after time t (and the Termination property of the algorithm is violated) or a decision is taken on v, since the system is v-valent at time t. Since $msgs(v)$ are never received by a correct process, the No loss property of the algorithm is violated in the latter case.

As a consequence of this result, any indirect consensus algorithm needs to ensure that the relationship "v-valence $\Rightarrow$ v-stability" for any configuration holds. This relationship between v-valence and v-stability is not trivially satisfied by a consensus algorithm.

3.2 Adapting Chandra-Toueg's $\Diamond\mathcal{S}$ consensus algorithm

The following paragraphs present the modification of the CT $\Diamond\mathcal{S}$ consensus algorithm [2] into an indirect consensus algorithm. First of all, a brief overview of the original $\Diamond\mathcal{S}$ consensus algorithm is presented. Then, the necessary modification to that algorithm and v-valence and v-stability are discussed. Finally, the adapted indirect consensus algorithm is presented and proved.

3.2.1 Chandra-Toueg's $\Diamond\mathcal{S}$ consensus algorithm

In [2], the authors present a consensus algorithm based on the unreliable failure detector $\Diamond\mathcal{S}$. The algorithm proceeds in rounds and requires a majority ($f < \frac{n}{2}$) of correct processes. It behaves as follows: at the beginning of each round, each process sends its estimate of the decision to the process acting as a coordinator in that round. The coordinator waits for a majority of estimates and selects the most recent one (based on its timestamp) and sends it to all processes. At this point, each process either receives the coordinator's proposal, or suspects the coordinator of having crashed. In the former case, the process sets its own estimate to the coordinator's proposal, updates its timestamp and sends a positive acknowledgement (*ack*) to the coordinator. In the latter case, a negative acknowledgement (*nack*) is sent. In both cases, the non-coordinator processes proceed to the next round.

The coordinator waits for a majority ($f + 1$) of answers. If all answers are *ack*s, the coordinator decides and informs the other processes of its decision. If at least one *nack* is received, the coordinator proceeds to the next round without deciding. It is easy to show that if $\lceil \frac{n+1}{2} \rceil$ of processes have accepted the coordinator's proposal v, then the system is in a v-valent configuration (i.e. any future decision is v), although the decision on v might only be taken in a later round.

3.2.2 Adapting the algorithm into an indirect consensus algorithm

In the original CT algorithm, a process that receives the coordinator's proposal in a given round updates its own estimate to match the proposal of the coordinator (and sends an *ack*). This is precisely the operation that allows the incorrect scenario presented in Section 2.2 to occur. Indeed, if all processes blindly adopt the coordinator's proposal v (thus leading to a v-valent configuration, with v a set of message identifiers) and that the originator of $msgs(v)$ crashes, then $msgs(v)$ might be lost and no v-stable configuration of the system can be reached.

In order to avoid this situation, we propose the following modification: whenever a process receives the coordinator's proposal v, it checks if $msgs(v)$ have been received (using the rcv function). If so, an *ack* is sent to the coordinator (the proposal is accepted); otherwise, a *nack* is sent (the proposal is refused). Similar approaches have been taken in [1, 7].

The modified algorithm The pseudo-code of the adapted indirect consensus algorithm is shown in Algorithm 2 (the parts that were modified with respect to the original CT algorithm have bold line numbers) and is expressed as in [9].

The lines 25 to 30 correspond to the modification described above. The rcv function is called at line 25 to test if all messages whose identifiers are in the coordinator's proposal have been received. The additional variable $estimate_c$ (lines 2, 18, 20, 21 and 37) represents the coordinator's proposal and can be different from the coordinator's own estimate $estimate_p$ (in case the coordinator does not have the messages corresponding to the estimate v with the highest timestamp). This is explained in the next paragraph.

The need for $estimate_c$ and $estimate_p$ Consider a coordinator c at line 21 that sends v to all in round 1 (c has received $msgs(v)$), and a process p_i that accepts this estimate at line 25 (p_i has received $msgs(v)$). In round 2, the coordinator c', if it receives the estimate from c or p_i selects it, even if it has not received $msgs(v)$. However, if c and p_i crash later, and no other process has received $msgs(v)$, no correct process might ever receive $msgs(v)$. So, in round 2 the coordinator c' updates $estimate_c$ with v, but $estimate_p$ is still equal to a different value. Once c and p_i crash, the estimate v with timestamp 1 will disappear, and an estimate with timestamp 0 will again be chosen.

This scenario illustrates that a process, including the coordinator, only accepts to modify its estimate if it has all the messages corresponding to the identifiers in the

Algorithm 2 Chandra-Toueg based $\Diamond\mathcal{S}$ indirect consensus algorithm (code of process p)

1: **procedure** $propose(v_p, rcv)$
2: $estimate_p \leftarrow v_p$, $estimate_c \leftarrow \perp$ {*p's and the coordinator's estimate of the decision value*}
3: $state_p \leftarrow undecided$
4: $r_p \leftarrow 0$ {r_p *is p's current round number*}
5: $ts_p \leftarrow 0$ {ts_p *is the last round in which p updated* $estimate_p$ }
6: **while** $state_p = undecided$ **do** {*rotate through coordinators until decision reached*}
7: $r_p \leftarrow r_p + 1$
8: $c_p \leftarrow (r_p \bmod n) + 1$ {c_p *is the current coordinator*}
9: **Phase 1:** {*all processes p send* $estimate_p$ *to the current coordinator*}
10: **if** $r_p > 1$ **then**
11: send $(p, r_p, estimate_p, ts_p)$ to c_p
12: **Phase 2:** {*coordinator gathers* $\lceil \frac{n+1}{2} \rceil$ *estimates and proposes new estimate*}
13: **if** $p = c_p$ **then**
14: **if** $r_p > 1$ **then**
15: **wait until** [for $\lceil \frac{n+1}{2} \rceil$ processes q : received $(q, r_p, estimate_q, ts_q)$ from q]
16: $msgs_p[r_p] \leftarrow \{(q, r_p, estimate_q, ts_q) \mid p$ received $(q, r_p, estimate_q, ts_q)$ from $q\}$
17: $t \leftarrow$ largest ts_q such that $(q, r_p, estimate_q, ts_q) \in msgs_p[r_p]$
18: $estimate_c \leftarrow$ select one $estimate_q$ such that $(q, r_p, estimate_q, t) \in msgs_p[r_p]$
19: **else**
20: $estimate_c \leftarrow estimate_p$ {*In the first round, the coordinator selects its own estimate*}
21: send $(p, r_p, estimate_c)$ to all
22: **Phase 3:** {*all processes wait for new estimate proposed by current coordinator*}
23: **wait until** [received $(c_p, r_p, estimate_{c_p})$ from c_p **or** $c_p \in \mathcal{D}_p$] {*query failure detector* $\mathcal{D}_p$}
24: **if** [received $(c_p, r_p, estimate_{c_p})$ from c_p] **then** {p *received* $estimate_{c_p}$ *from* c_p}
25: **if** $rcv(estimate_{c_p})$ **then** {*check if all messages in* $estimate_{c_p}$ *have been received*}
26: $estimate_p \leftarrow estimate_{c_p}$
27: $ts_p \leftarrow r_p$
28: send (p, r_p, ack) to c_p
29: **else** {p *received an estimate* v *from the coordinator but* V *is missing*}
30: send $(p, r_p, nack)$ to c_p
31: **else** {p *suspects that* c_p *crashed*}
32: send $(p, r_p, nack)$ to c_p
33: **Phase 4:** {*the coordinator waits for* $\lceil \frac{n+1}{2} \rceil$ *replies. If all replies adopt its estimate, the coordinator R-broadcasts a decide message*}
34: **if** $p = c_p$ **then**
35: **wait until** [for $\lceil \frac{n+1}{2} \rceil$ processes q : received (q, r_p, ack) **or** for 1 process q: $(q, r_p, nack)$]
36: **if** [for $\lceil \frac{n+1}{2} \rceil$ processes q : received (q, r_p, ack)] **then**
37: *R-broadcast* $(p, estimate_c, decide)$ to all
38: **when** R-deliver $(q, estimate_q, decide)$ {*if p R-delivers a decide message, p decides accordingly.*}
39: **if** $state_p = undecided$ **then**
40: ***decide****(*$estimate_q$*)*
41: $state_p \leftarrow decided$

new estimate. Since only non-crashed processes send their estimates to the coordinator at the beginning of each round, eventually, only estimates adopted by at least one correct process are received by the coordinator.

3.2.3 Proof of the algorithm (sketch)

The Uniform integrity and Uniform validity properties are trivially proven and are not shown here. The proof of Uniform agreement is presented in [3] and is almost identical to the proof in [2].

Termination There is a time t such that all faulty processes have crashed. After this time t, all correct processes have an estimate v such that $rcv(v)$ holds. From *Hypothesis A*, there is thus a time t' such that $rcv(v)$ holds for all correct processes and for the estimate v of any correct process. After this time t', the indirect consensus algorithm behaves exactly like the original consensus algorithm. Thus, if a decision hasn't been taken before t', the Termination property of Chandra-Toueg's $\Diamond\mathcal{S}$ consensus algorithm guarantees that the indirect consensus algorithm terminates.

No loss We show that any v-valent configuration is also v-stable. If a configuration is v-valent, it implies that the coordinator always selects v as its proposal. Since the proposal selected by the coordinator is one of the $\lceil \frac{n+1}{2} \rceil$ received estimates, and that the coordinator always receives v at least once, at least a majority of processes have an estimate equal to v.

These $\lceil \frac{n+1}{2} \rceil$ processes that have an estimate equal

to v can either (1) have started consensus with v or (2) adopted the proposal v of a previous coordinator, in which case the rcv function on v is verified. In both cases, $msgs(v)$ have been received by a majority of processes and the configuration is v-stable. Since any v-valent configuration is also v-stable, the No loss property holds.

3.3 Adapting Mostéfaoui-Raynal's $\Diamond\mathcal{S}$ consensus algorithm

We start by presenting an informal overview of the original MR consensus algorithm. The problems encountered when adapting this algorithm into an indirect consensus algorithm are then discussed. The solution to these problems, which modifies the resilience of the algorithm, is then presented. Finally, the adapted indirect consensus algorithm is presented and proven correct.

3.3.1 Mostéfaoui-Raynal's $\Diamond\mathcal{S}$ consensus algorithm

In [8], the authors present a consensus algorithm based on unreliable failure detectors and quorums. We consider their $\Diamond\mathcal{S}$ based algorithm here. As in the CT consensus algorithm, the MR algorithm proceeds in rounds and requires a majority ($f < \frac{n}{2}$) of correct processes. In rounds without failures and without suspicions, a decision can be taken within two communication steps by all processes.

Each round consists of two phases. At the beginning of Phase 1, the coordinator of that round sends its estimate to all processes. Each process then either receives the coordinator's proposal, or suspects the coordinator of having crashed. In the latter case, the process considers that an invalid value ($\perp$) was received from the coordinator. In both cases, the process sends the estimate received from the coordinator (a valid value or $\perp$) to all processes, which concludes Phase 1 of the algorithm.

In Phase 2, each process waits for a majority of estimates (including the one possibly received from the coordinator). If all received estimates are the same value v, the process decides v and informs all other processes of its decision. If this is not the case, but at least one received estimate is valid (not $\perp$), the process sets its own estimate to the received valid estimate and proceeds to the next round.

The Uniform agreement property of consensus is ensured by the fact that if a decision on v is taken by a process p, then p has received the estimate v from $\lceil \frac{n+1}{2} \rceil$ processes. This in turn ensures that all processes have received at least one estimate equal to v and have thus set their own estimate to v. Since the estimates of all processes are equal to v, any subsequent decision can only be done on v.

3.3.2 Problems adapting the Mostéfaoui-Raynal consensus algorithm into an indirect consensus algorithm

As described above, one of the constraints for Uniform agreement to hold in the MR consensus algorithm is that any process that receives at least one valid estimate must accept that estimate, i.e. modify its own estimate to match the received one. Accepting such an estimate might however lead to a violation of the No loss property of indirect consensus. This is shown by two executions that are indistinguishable for some process p: in one execution the configuration is v-valent but not v-stable; in the other execution the configuration is v-valent and v-stable.

We assume a system with n processes and p a non-coordinator process in the current round of the algorithm. Process p suspects the coordinator and p does not have the messages corresponding to the coordinator's proposal v. The two executions are the following:

(1) the coordinator is correct. $\lceil \frac{n+1}{2} \rceil$ processes accept the proposal of the coordinator, whereas $\lfloor \frac{n-1}{2} \rfloor$ suspect the coordinator. The coordinator receives $\lceil \frac{n+1}{2} \rceil$ estimates equal to its own proposal whereas p receives one estimate equal to the coordinator's proposal and $\lfloor \frac{n-1}{2} \rfloor$ invalid $\perp$ values. In this execution, the coordinator decides. To guarantee the Uniform agreement property of consensus, p must accept the coordinator's proposal v (and thus modify its own estimate), although it doesn't have $msgs(v)$.

(2) the coordinator is faulty. Let us assume that the $n-1$ non-coordinator processes suspect the coordinator and do not have the messages corresponding to the coordinator's proposal v. They thus all send a $\perp$ value. Process p receives the coordinator's proposal as well as $\lfloor \frac{n-1}{2} \rfloor$ invalid $\perp$ values. If the coordinator crashes before any process receives $msgs(v)$, then $msgs(v)$ might be lost. In this execution, p must not accept the coordinator's proposal v.

If p takes a conservative approach and only accepts a proposal v if it has $msgs(v)$ (or that at least one correct process has $msgs(v)$), then the Uniform agreement property would be violated in the first execution. If, on the other hand, p takes the optimistic approach of accepting a proposal v even if it doesn't have $msgs(v)$, this could lead to a v-valent configuration that is not v-stable. The No loss property of indirect consensus could thus be violated. Therefore, any of the approaches that the algorithm chooses to implement leads to the violation of one of the indirect consensus properties.

The modifications must thus ensure both of the following properties: (i) a process should only accept v if it has $msgs(v)$ or $f+1$ processes have $msgs(v)$; (ii) if a process decides v in round r, then all non-crashed processes

must adopt v during round r. Property (i) and (ii) aim at guaranteeing No loss, respectively Uniform agreement.

3.3.3 Modified Mostéfaoui-Raynal algorithm

Consequences on the resilience Thus, we must modify the MR algorithm. In Phase 1 of the algorithm, a process can only accept the coordinator's proposal v if it has received $msgs(v)$ (at this point, a process does not know if any other process has adopted v and can therefore not know if v is stable, which is the second possible condition for adopting v). Thus, at the end of Phase 1, when a process p sends the estimate v to all processes, this estimate is a non-$\perp$ value only if p has received $msgs(v)$.

In Phase 2 of the consensus algorithm, all processes wait for $n - f$ estimates from the other processes. If all of these estimates are identical, a decision can be taken. If they are not, a process p can accept a valid estimate v if (1) p has received $msgs(v)$ or (2) if the estimate v was received from at least $f + 1$ processes (i.e. from at least one correct process that has received $msgs(v)$).

To ensure Uniform agreement, we have seen that if a decision is taken on v, then all processes must accept v as their own estimate. Not all processes have necessarily received $msgs(v)$ (which means that condition (1) above might not be true for all processes). Therefore, if a decision is taken, the algorithm must ensure that the condition (2) above is true for all processes (i.e. all processes receive at least $f + 1$ estimates equal to v). This can be ensured as follows.

Each process waits for $n-f$ estimates at the beginning of Phase 2. Since there are at most n estimates in the system, each pair of processes receives a common set of estimates. The minimum size of this common set is $n - 2f$ (assuming $f < \frac{n}{2}$). So condition (2) is ensured if $n - 2f \geq f + 1$, which leads to $f < \frac{n}{3}$.

The modified MR algorithm The pseudo-code of the adapted MR indirect consensus algorithm is shown in Algorithm 3 (the parts that were modified with respect to the original algorithm have bold line numbers) and is expressed as in [9].

The lines 16 to 19 correspond to the modifications to the first phase of the algorithm. With these modifications, a process accepts the coordinator's proposal v only if it received $msgs(v)$ (in the original consensus algorithm, v was always accepted). In Phase 2, the modifications are two-fold. First of all, the condition $f < \frac{n}{3}$ force each process to wait for $\lceil \frac{2n+1}{3} \rceil$ estimates at the beginning of Phase 2 (lines 21 and 22). Secondly, if a process receives a valid estimate v as well as $\perp$ values, the valid estimate is adopted if (1) the process has $msgs(v)$ or (2) if the estimate v was received from more than one third of the processes (lines 28 and 29).

The remaining parts of the indirect consensus algorithm are identical to the original MR consensus algorithm.

3.3.4 Proof of the algorithm (sketch)

The Uniform integrity and Uniform validity properties are trivially proven and are therefore not shown here. The proof of Termination is similar to the Termination proof in Section 3.2.3 and can be found in [3].

Uniform agreement Let process p be the first process that reliably broadcasts a decision message and then decides on some value v. Process p previously received the estimate v from $\lceil \frac{2n+1}{3} \rceil$ processes in Phase 2 of a given round r. All other processes also received $\lceil \frac{2n+1}{3} \rceil$ values in round r and thus received at least $\lceil \frac{n+1}{3} \rceil$ identical values to p. Thus, all processes eventually execute line 25 or 29 in round r and set their own estimate to v. After round r, the estimate of all processes is thus equal to v.

No loss In the modified Mostéfaoui-Raynal indirect consensus algorithm, a system is in a v-valent configuration if $\lceil \frac{2n+1}{3} \rceil$ processes accept the same estimate v in a given round r. The estimate of a process is equal to v in three cases : (1) consensus was executed with v as the initial proposal, (2) the process received v in Phase 1 or 2 and accepted it because $msgs(v)$ had been previously received or (3) the process received v in Phase 2 from at least $f + 1$ processes. Since at least $\lceil \frac{2n+1}{3} \rceil$ processes have the same estimate v in the v-valent configuration, at least $f + 1$ processes must have modified their estimate following cases (1) or (2) above. In both of these cases, the processes have $msgs(v)$. The configuration is thus v-stable, since at least $f + 1$ processes have $msgs(v)$. Since any v-valent configuration is also v-stable, the No loss property is verified.

4 Performance measurements

In Section 2, we presented a short performance comparison between atomic broadcast with consensus on messages and consensus on message identifiers. In the following paragraphs, we present measurements comparing indirect consensus to (the faulty implementation of) consensus directly on message identifiers, in order to estimate the overhead introduced by the indirect consensus solution. This section starts by a presentation of the system setup and the Neko framework that was used in the

Algorithm 3 Mostéfaoui-Raynal based $\Diamond\mathcal{S}$ indirect consensus algorithm (code of process p)

```
1: procedure propose(v_p, rcv)
2:     estimate_p ← v_p                                   {estimate_p is p's estimate of the decision value}
3:     state_p ← undecided
4:     r_p ← 0                                            {r_p is p's current round number}
5:     while state_p = undecided do                       {rotate through coordinators until decision reached}
6:         r_p ← r_p + 1
7:         c_p ← (r_p mod n) + 1                          {c_p is the current coordinator}
8:         est_from_c_p ←⊥                                {est_from_c_p is the estimate received from the coordinator or invalid (⊥)}
9:         Phase 1:                                       {coordinator proposes new estimate; other processes wait for this new estimate}
10:            if p = c_p then
11:                est_from_c_p ← estimate_p
12:                send (p, r_p, est_from_c_p) to all
13:            else
14:                wait until [received (c_p, r_p, est_from_c_{c_p}) from c_p or c_p ∈ D_p]   {query failure detector D_p}
15:                if [received (c_p, r_p, est_from_c_{c_p}) from c_p] then                   {p received est_from_c_{c_p} from c_p}
16:                    if [rcv(est_from_c_{c_p})] then
17:                        est_from_c_p ← est_from_c_{c_p}
18:                    else
19:                        est_from_c_p ←⊥
20:                send (p, r_p, est_from_c_p) to all
21:        Phase 2:        {each process waits for ⌈(2n+1)/3⌉ replies. If they indicate that ⌈(2n+1)/3⌉ processes adopted the proposal, the process
           R-broadcasts a decide message}
22:            wait until [for ⌈(2n+1)/3⌉ processes q : received (q, r_p, est_from_c_q)]
23:            rec_p ← {(q, r_p, est_from_c_q) | p received (q, r_p, est_from_c_q) from q}
24:            if rec_p = {v} then
25:                estimate_p ← v
26:                R-broadcast (p, estimate_p, decide)                                       {R-broadcast without the initial send to all}
27:            else if rec_p = {v, ⊥} then                                                   {accept v if (1) rcv(v) is true or (2) v was received ⌈(n+1)/3⌉ times}
28:                if rcv(v) or |{p received (q, r_p, v) from q}| ≥ ⌈(n+1)/3⌉ then
29:                    estimate_p ← v
30: when R-deliver (q, estimate_q, decide)                                                   {if p R-delivers a decide message, p decides accordingly}
31:     if state_p = undecided then
32:         decide(estimate_q)
33:         state_p ← decided
```

experiments. Several comparisons between indirect consensus and the faulty implementation using consensus are then presented.

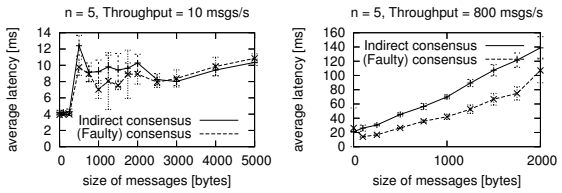
System setup and the Neko framework The benchmarks were run on a cluster of PCs running Red Hat Linux (kernel 2.4.18). The PCs have Pentium III 766 MHz processors and 128 MB of RAM, and are interconnected by a 100 Base-TX Ethernet. The Java Virtual Machine was Sun's JDK 1.4.1_01.

Neko [10] is a simulation and prototyping framework. Using this framework, the same (Java) implementation of a protocol can be used in a simulated environment and on a real network. The protocols are implemented as layers of a stack. The CT atomic broadcast algorithm was implemented and executed either on messages or on message identifiers, according to the test configuration. The indirect consensus algorithm was implemented based on an already existing implementation of the CT $\Diamond\mathcal{S}$ consensus algorithm that was used in previous performance studies [12, 11]. All the results presented here were obtained on the real network described above.

Performance metric: latency versus message size The performance metric for atomic broadcast is the *latency*, defined as the average (over all processes) of the elapsed time between *abroadcasting* a message m and *adelivering* m. A simple symmetric workload is used: all processes *abroadcast* messages at the same rate and the global rate is called the *throughput*.

To quantify the overhead introduced by indirect consensus compared to (the faulty implementation of) consensus directly on message identifiers, we present figures showing the latency of atomic broadcast as a function of the message size, for low and high throughputs. Performance results for the latency as a function of the throughput (for a given message size) can be found in [3].

Performance results: overhead of indirect consensus Figure 2 compares the performance of indirect consensus and consensus directly on message identifiers as the size of the messages increases. The overhead ratio re-



(a) 10 msgs/s (b) 800 msgs/s

Figure 2. Latency vs. payload of the atomic broadcast algorithm using indirect consensus or (faulty) consensus on message identifiers in a system with 5 processes

mains stable as the size of the messages varies. In the case of low throughputs (10 messages per second), the overhead is negligible for all message sizes. For higher throughputs, the overhead is clearly measurable, but does not vary much as the size of the messages increases. These results are expected: since both algorithms only use the message identifiers to reach a decision, the messages themselves (and thus their size) do not affect the performance of the indirect consensus and consensus algorithms.

Discussion In Section 2, we saw that executing atomic broadcast using indirect consensus (on message identifiers) provides better performance than using consensus on messages, especially as the sizes of the system or the messages increase. In previous group communication stack implementations, consensus was often executed directly on message identifiers, which can lead to faulty executions if a process crashes. The indirect consensus approach, which solves this problem, yields performance results that are comparable to the faulty solution (in the case of an indirect consensus algorithm with the same degree of resilience as the corresponding consensus algorithm), as Figure 2 and the additional results in [3] show. The cost of adopting a correct implementation of atomic broadcast on message identifiers is thus fairly low and ensures that the properties of atomic broadcast hold, even if processes crash.

5 Conclusion

In [2], atomic broadcast is reduced to consensus on messages. This reduction is correct, but since consensus is executed on sets of messages, it yields poor performance as the size of the messages increases. Instead, consensus can be executed on message identifiers, which decouples the consensus algorithm from the size of the messages. This can however lead to the violation of the Validity property of atomic broadcast. *Indirect* consensus addresses this issue by providing a *No loss* property, which guarantees that all messages whose identifiers have been decided upon are eventually delivered by atomic broadcast. To ensure the No loss property, the indirect consensus algorithm must guarantee that any v-valent configuration (any future decision is v) is also v-stable (at least one correct process has received the messages whose identifiers are in v).

The paper has shown that adapting a consensus algorithm into an indirect consensus algorithm is not trivial. The resilience of the adapted Mostéfaoui-Raynal $\Diamond\mathcal{S}$-based indirect consensus algorithm is $f < \frac{n}{3}$ whereas the original consensus algorithm supports $f < \frac{n}{2}$ failures. Chandra-Toueg's $\Diamond\mathcal{S}$-based consensus algorithm does not have this problem and was easy to adapt.

Finally, the performance of the Chandra-Toueg based indirect consensus algorithm is better than the original consensus algorithm on messages and comparable to the performance of the faulty implementation of the consensus algorithm directly on message identifiers.

References

[1] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI: Symposium on Operating Systems Design and Implementation*. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999.

[2] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2):225–267, 1996.

[3] R. Ekwall and A. Schiper. Solving atomic broadcast with indirect consensus. Technical Report LSR-REPORT-2006-001, Switzerland, 2006.

[4] P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998. Number 1867.

[5] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of ACM*, 32(2):374–382, Apr. 1985.

[6] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. TR 94-1425, Dept. of Computer Science, Cornell University, Ithaca, NY, USA, May 1994.

[7] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[8] A. Mostéfaoui and M. Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors: A general quorum-based approach. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC)*, number 1693 in Lecture Notes in Computer Science, pages 49–63, Bratislava, Slovak Republic, Sept. 1999. Springer-Verlag.

[9] P. Urbán. *Evaluating the Performance of Distributed Agreement Algorithms: Tools, Methodology and Case Studies*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, Aug. 2003. Number 2824.

[10] P. Urbán, X. Défago, and A. Schiper. Neko: A single environment to simulate and prototype distributed algorithms. *Journal of Information Science and Engineering*, 18(6):981–997, Nov. 2002.

[11] P. Urbán, N. Hayashibara, A. Schiper, and T. Katayama. Performance comparison of a rotating coordinator and a leader based consensus algorithm. In *Proc. 23nd IEEE Int'l Symp. on Reliable Distributed Systems (SRDS)*, pages 4–17, Florianópolis, Brazil, October 2004.

[12] P. Urbán, I. Shnayderman, and A. Schiper. Comparison of failure detectors and group membership: Performance study of two atomic broadcast algorithms. In *Proc. of the Int'l Conf. on Dependable Systems and Networks (DSN)*, pages 645–654, June 2003.